XTI
An Extended Type Information Library

Bjarne Stroustrup

AT&T Labs – Research

http://www.research.att.com/~bs

Abstract

This talk presents a library representing types, names, and declarations of a C++ program, part of a C++ program, or even several C++ programs so that they can be easily analyzed and manipulated from within a C++ program. Basically, XTI presents an abstract tree representation of C++ and provides ways of traversing, querying, and annotating that representation. So far, this library has been used to generate distributed programs from Standard C++ source. It's original purpose was the support of distributed programming. Another application has been a library for reading and writing an XML representation of C++.

60 minutes

Overview

- The Original Project
- General ideal and ideas
- Three traversal
  - External polymorphism and visitors
- Memory management
- Hierarchy design
  - Abstract and concrete classes
- Scope: The central abstraction
- Classes, interfaces, operations
- Final thoughts

Work in Progress

- I'm still modifying the code in significant ways
  - So far, only two (experimenting) users
  - Not every part of the library has been completed
- That means
  - Lack of use
  - Lack of experience
  - Design errors
  - Bugs
- Opportunities for improvements
  - Focus on design decisions

The Original Project

- Communication with remote mobile device
  - Calling interface
    - CORBA, DCOM, Java RMI, …, homebrew interface
  - Transport
    - TCP/IP, XML, …, homebrew protocol

- Big, Ugly, Slow, Proprietary, …
  - Why can’t I just write ISO Standard C++?

The original Project
Distributed programs in ISO C++

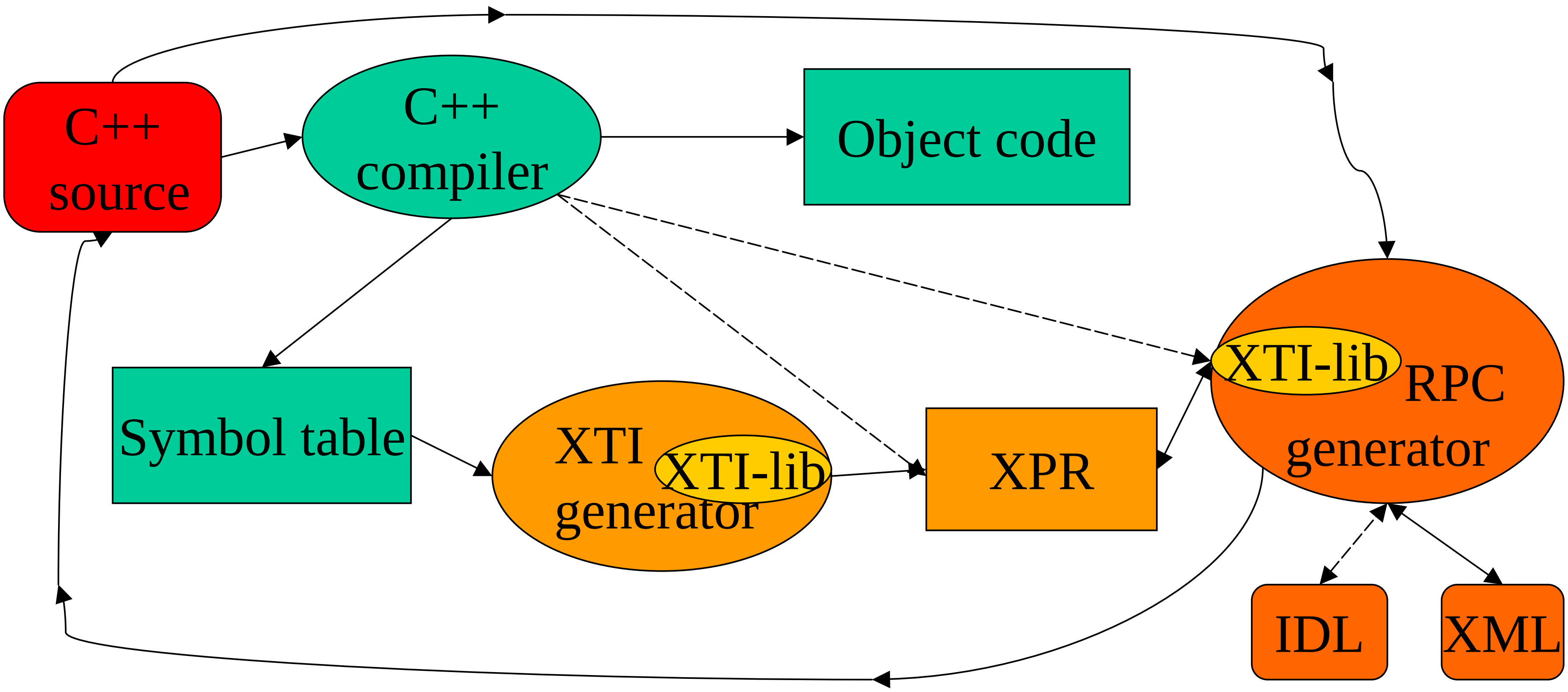
```
// use local object:


X x;
A a;
std::string s("abc");
// …
x.f(a, s);
```

```
// use remote object :
remote_proxy<X> x;
x.connect("my_host");
A a;
std::string s("abc");
// …
x.f(a, s);
```

- “as similar as possible to non-distributed programming, but no more similar”

Program generation/transformation



Classical OO approach/solution

- We need to model
  - The C++ type system
  - Eventually all of C++ (or almost all)
- Open ended problem
- In memory and persistent representation
  - Ease of programming essential
  - Extensibility essential
- Run-time resolution of operations essential
  - A representation is read from input, possibly incrementally
  - Virtual functions essential for simple programming
- Long-lived system
  - Complete information hiding essential
- Many implementations likely
  - Data representations will differ
  - Interfaces must be common

XTI
eXtended Type Information

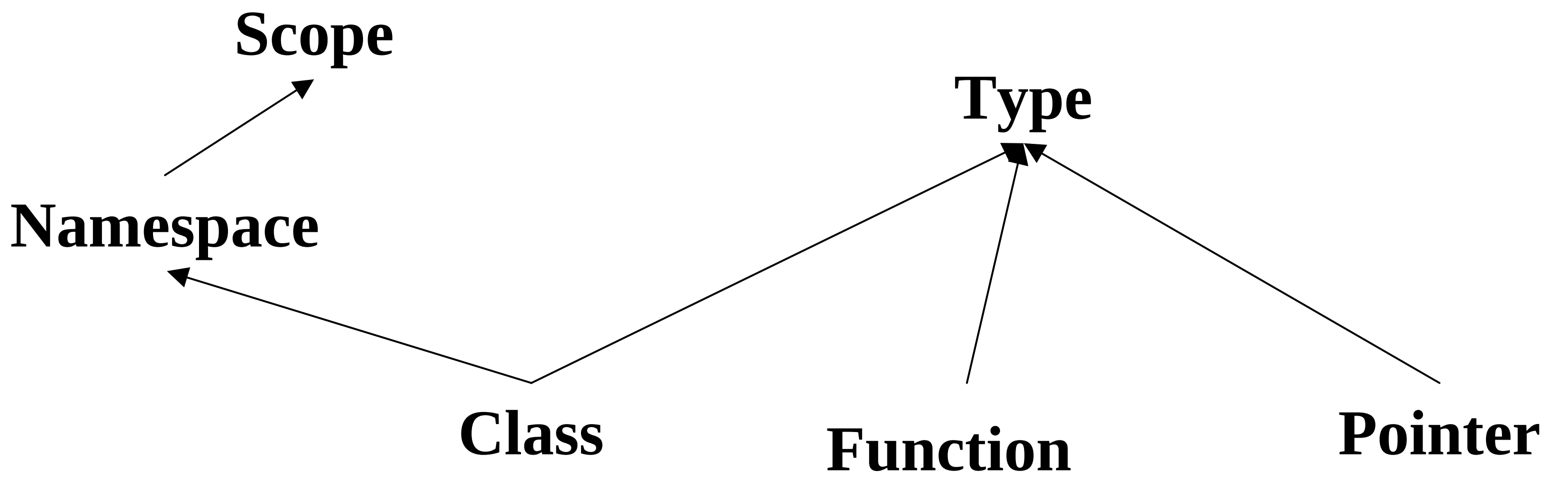
- Library written in Standard C++
  - Easy/simple
    - lookup based on name
    - traversal of scope
    - query based on attribute
    - “call interface”
  - Buzzword compliant
    - object-oriented, generic, abstract, etc.
    - Small, fast, complete, portable, etc.
    - Easy to learn (we hope)

XTI Ideals (Design principles)

- Direct representation of C++'s complete type system as classes
  - Built-in types, classes, templates, …
  - Can represent erroneous and incomplete C++ programs
- Programming effort proportional to complexity of task
  - XTI is not just a data structure
- Hide the representation
  - define the semantics, not the representation
- The set of operations on the XTI must be extensible
- Minimal overhead (run-time and space)
  - XTI is produced/read only when needed
- No integration with compiler
  - No language feature must depend on XTI

Basic Idea

Classical hierarchy of types, etc.:



Each node type will provide operations useful for manipulating that kind of C++ construct, e.g., a Function will hold information about number of arguments, type of arguments, etc.

XTI

- A set of classes/objects representing C++ declarations:

```
Program prog("my_types");
if (prog.global_scope["My_vec"].is_class()) { /* … */ }
for (scope::iterator p = prog.begin(); p!=prog.end(); ++p) p->xti_name();
```

- Represent anything that the C++ type system can
  - Classes, enumerations, typedefs, templates, namespaces, functions, non-local variables
  - Not (yet) code, local types, local variables

XTI: eXtended Type Information

- Use for program analysis
  - E.g. ODR verification, version consistency checking
- Use for program transformation
  - E.g. IDL generation
- Use for run-time resolution
  - “Calling interface”
    - Function call
    - Object creation

External Polymorphism

- A tree representing the C++ type system will have dozens of types of nodes
  - Could have hundreds
- “climbing a tree” with dozens of node types using **if-then-else** and **switch** statements is tedious and error prone
  - That’s why we use virtual functions
- A user cannot add new virtual functions
  - Visitor, or
  - Simulate virtual functions (switch hidden in library template)

XTI_visitor

```
struct XTI_obj {
    // …
    virtual void accept(XTI_visitor&) const;    // hook for visitor pattern
};

Struct XTI_visitor {
    virtual void visit(Const Class&) = 0;
    virtual void visit(Const Function&)=0;
    // …
};

Struct My_visitor : XTI_visitor {
    // my data (if any)
    void visit(Const Class& c) { /* what I want done for a class */ }
    void visit(Const Function& f) { /* what I want done for a function */ }
    // …
};
```

XPR
eXternal Program Representation

- Easy/fast to parse
- Easy/fast to write
- Can be thought of as a specialized portable object database
- Compact
  - About as compact as C++ source code
- Robust
  - Read/write without using a symbol table
- LR(1), strictly prefix declaration syntax
- Human readable
- Human writeable
- Can represent almost all of C++ directly
  - No preprocessor directives
  - No multiple declarators in a declaration
  - No <, >, >>, or << in template arguments, except in parentheses

XPR

```
i : int                              // int i;
C : class {                          // class C {
    m : const int                    //     const int m;
    mm : *const int                  //     const int* mm;
    f : (:int,:*char) double         //     double f(int,char*);
    f : (z:complex) C                //     C f(complex z);
}                                    // };
vector : <T> class {                 // template<class T> class vector {
    p : *T                           //     T* p;
    sz : int                         //     int sz;
}                                    // };
```

User-interface principles I

- Type safe
- Abstract
  - No representation dependencies
- No memory management required of users
- Separate interfaces for
  - Plain users: immutable type information
    - Const references
  - Providers of XTI
    - Non-const pointers

Memory management

- Initial idea: reference counted smart pointer
  - Works, but inelegant
    - Why should “ordinary users” see pointers at all?
  - Many pointers are to non-shared objects
    - inefficient
- Second idea: only some pointers counted
  - Too complex to be manageable
- Third idea: let XTI own all pointers
  - Put every pointer to an XTI object in a vector
    - Delete elements of that vector “at end”
  - Don’t give users pointers: const references
  - Not every object needs to be on the free store: member objects

XTI_obj

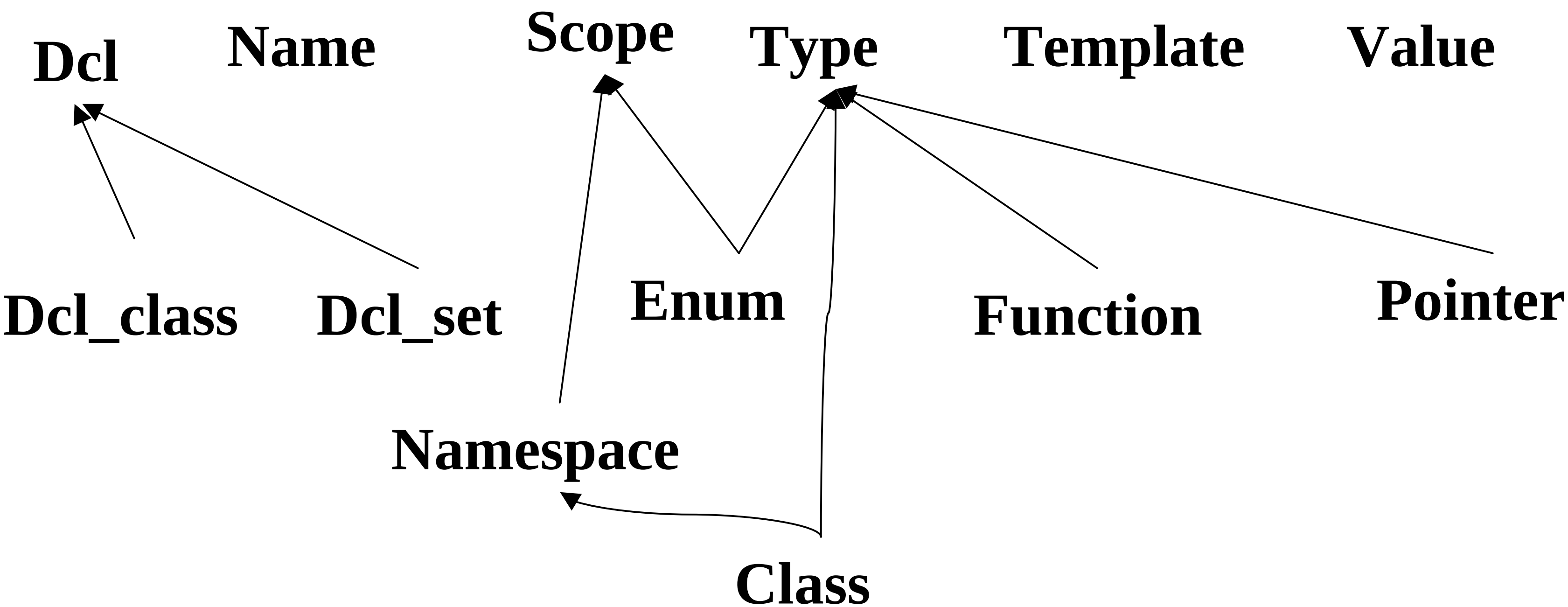
```
struct XTI_obj {
    Kind k;                               // placeholder for debugging support
    virtual  ~XTI_obj() { }               // the virtual destructor is essential
    virtual void accept(XTI_visitor&) const;     // hook for visitor pattern
};

class Program_impl {
    vector<XTI_obj*> owned;               // every XTI_objs is owned by a Program
    // …
    ~Program_impl() { /* delete all elements of owned */ }
    // …
    template<class T, class A> T* make(A a)     // make all XTI_objs using make()
    {
        T* p = new T(a);
        owned.push_back(p);
        return p;
    }
}
```

User-interface principles II

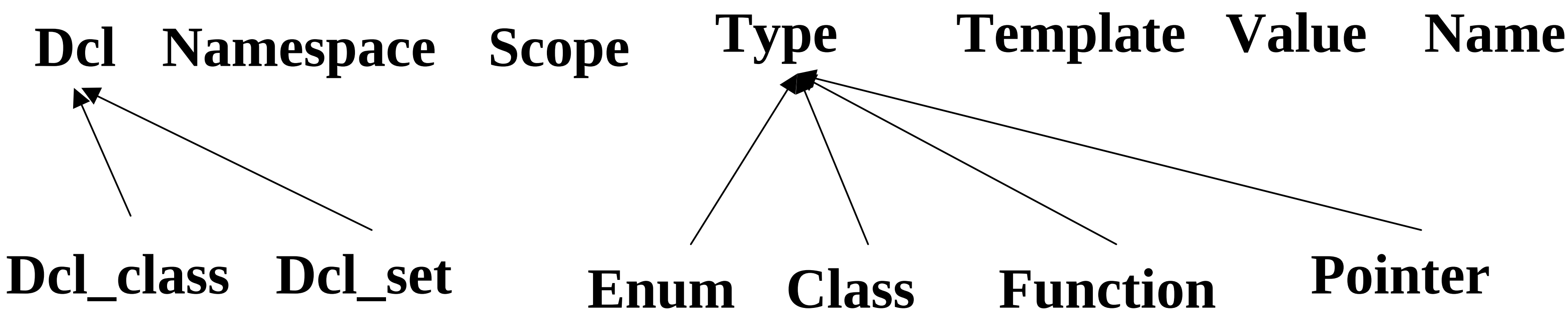
- Complete separation between
  - xti.h
    - Completely abstract
      - imagine multiple implementations, incremental build
    - For users of XTI
    - Scope, Class, Name, Pointer, …
    - References, iterators, const access only
      - But users can add attributes
  - concerte_xti.h
    - Encapsulated data
      - Concrete_xti has users: not just a struct
    - For builders of XTI
    - Scope_impl, Class_impl, Name_impl, Pointer_impl, …
    - Pointers, non-const access

XTI classes I



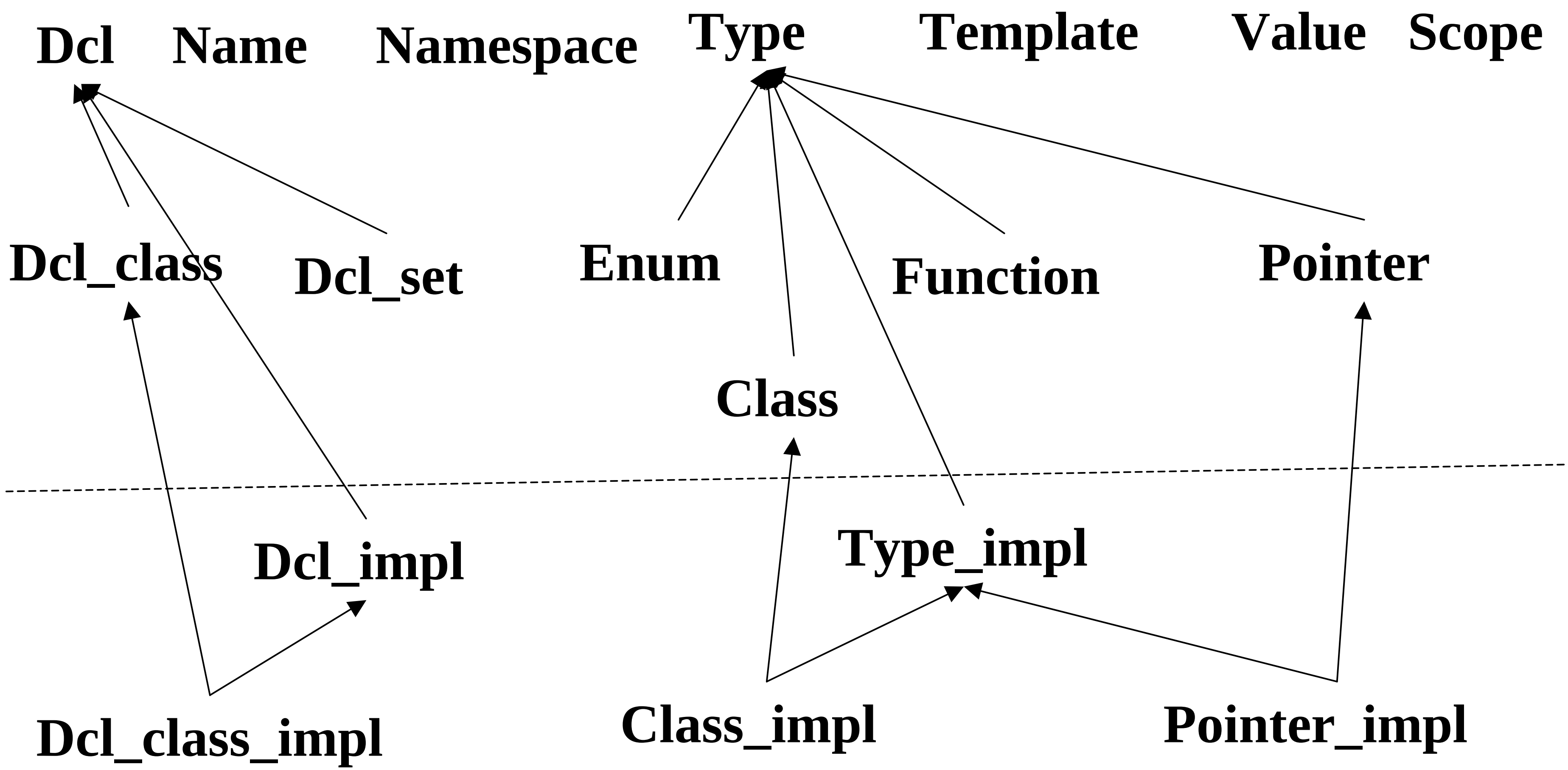
- A class is a namespace and a namespace is a scope
  – Beautiful code re-use
  – Messy class hierarchy, especially when implementations are considered

XTI classes II



- Simpler structure
  - a namespace has a scope, a class has a scope
- Needs a few forwarding functions
  - Still not quite as elegant as class -> namespace -> scope
- Initial change took 15 minutes, but I keep needing more forwarding functions

Concrete XTI classes



XTI classes

```
struct Type : public XTI_obj {      // common base class for all Types
    virtual Type_properties properties() const = 0;
    virtual string id() const = 0;

    bool is_const() const;              // convenience functions
    bool is_volatile() const;

    template<class T> bool is() const
        { return dynamic_cast<const T*> (this); }
    template<class T> const T& get() const
        { return dynamic_cast<const T&>(*this):
};
```

XTI classes

```
class Class : public virtual Type {
    virtual const Scope& bases() const = 0;
    virtual const Scope& members() const = 0;
    virtual const Name& get_name() const = 0; // full name (incl. scope)
    virtual string id() const = 0;                       // name string
    virtual void accept(XTI_visitor& v) const { v.visit(*this); }
};
```

Concrete XTI classes

```
class Class_impl : public Class, public Type_impl  {
    Scope_impl base;
    Scope_impl memb;
Public:
    Class_impl(Type_properties, Name_impl*);

    const Scope& bases() const;     // override class virtuals
    const Scope& members() const;
    const Name& get_name() const;
    string id() const;

    Scope_impl* base_impl();        // functions for building XTI
    Scope_impl* member_impl();
    const Name* name_impl() const;
    Name* name_impl();
};
```

Scope

- The central abstraction / simplification
  - Used for all sequences of (name,value) pairs
    - Global scope
    - Namespace
    - Class
    - Enumeration
    - Argument list
    - Template argument list
    - Annotation list
  - Access: Indexed, iterator, and name
    - All run-time checked (throw exceptions)
  - Order preserving

Iterators

- First idea
  - Maintain user/implementer distinction
    - Abstract class **Iter** (pure interface, holding no data**)**
    - Concrete class **Iter_impl** (carrying data, different representations can be used without affecting users)
- But that’s impossible
  - Users need to create and copy **Iter**s
- And it’s wrong (over-abstraction)
  - An random-access iterator is an abstraction for something holding 0,1,2,3,…
  - We don’t need an abstraction for an abstraction
- Obvious solution
  - **Iter** is a type holding values in the range [0,n)
    - Range checked for type safety

Class Scope

```
struct Scope {
    // define iterator type: basically range checked long
public:
    virtual const Dcl& operator[](const string&) const = 0;        // map style access
    virtual const Dcl& operator[](const char*) const = 0;          // v[0] ambiguous?

    virtual const Dcl& operator[](int) const = 0;                  // vector style access
    virtual size_t size() const = 0;

    virtual bool has_member(const string& s) const = 0; // convenience only
    virtual bool has_member(const char* s) const = 0;


    virtual Iter begin() const = 0;                     // STL container style access
    virtual Iter end() const = 0
    virtual Iter position(const string& s) = 0;
    virtual Iter position(const char* s) = 0;
    virtual Iter position(int i) = 0;

    virtual const string& name() const = 0;
    virtual const Name& get_name() const = 0;   // Name contains enclosing Scope
};
```

Class Scope_impl

```
class Scope_impl : public virtual Scope, public XTI_obj {
    // a scope maps strings (not Names) to Dcls: you can look up by variable name (string)
    // or by index (declarations are numbered [0,n), and you can iterate through a scope

    vector<Dcl_impl*> v;              // declarations in declaration order
    mutable map<string,int> m;        // map into v index
    Scope_impl* encl;                 // enclosing scope
    Name_impl* n;                     // optional name
    Program_impl* prog;               // containing program

    // …
};
```

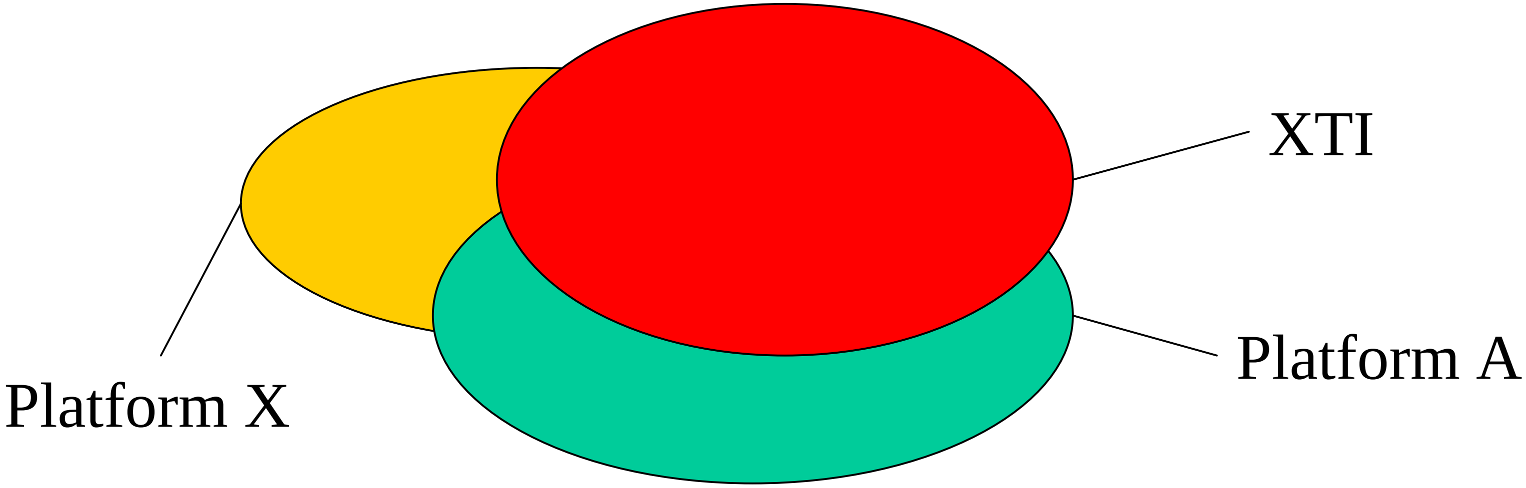
What should be represented?

- Answer any question that can be asked about declarations in source code after preprocessing
  - Typedefs, constants, templates, template specializations, access control, declaration order, variable names, …
  - Many semantically and syntactically erroneous programs
  - Later: expressions, function bodies
  - Probably not: line numbers, file names, comments (annotations)
- **Program** roughly equivalent to translation unit
  - But a user can write a program to merge external XTI files
- Allow addition of “annotations”
  - Sizeof, offset
  - Line numbers, file names
  - By compiler, by XTI writer, and by programmer

What can/cannot be done with XTI?

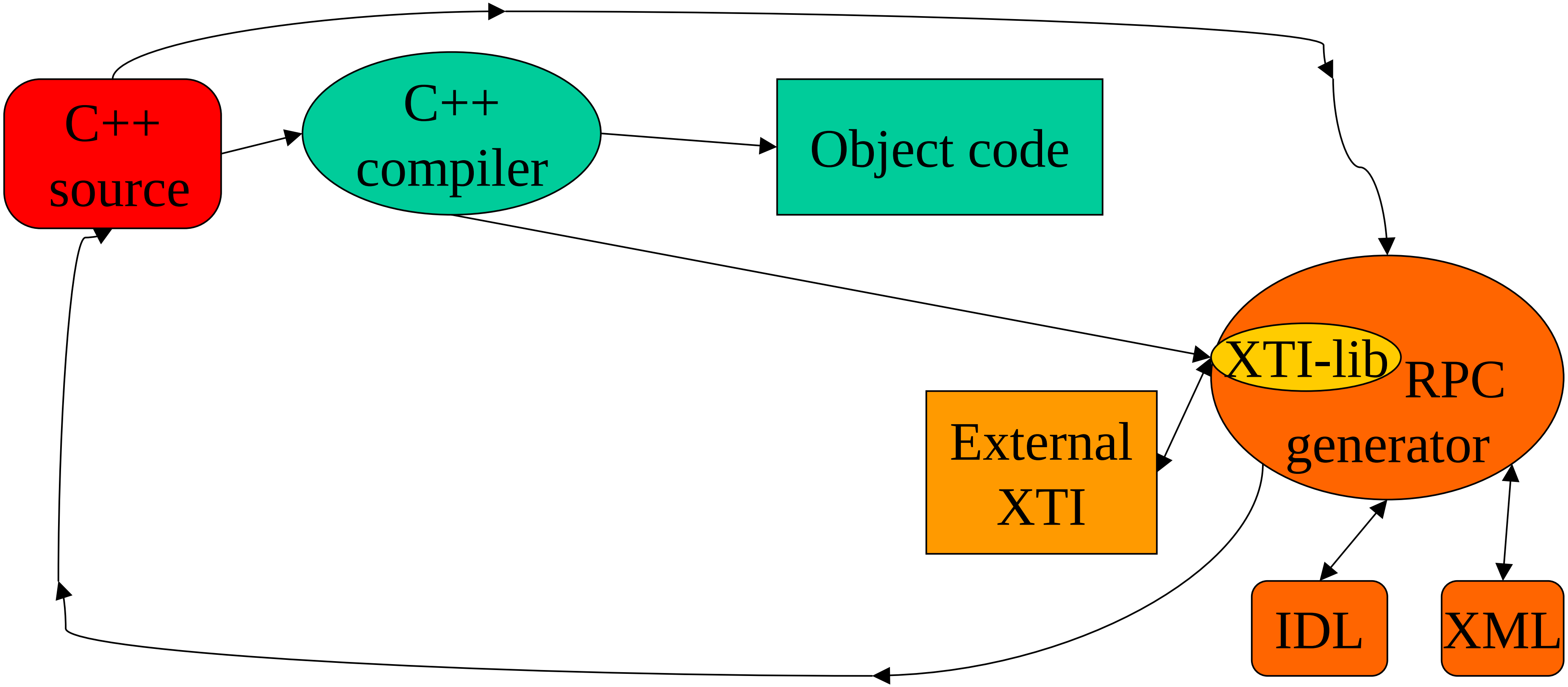
- Read in XTI for the running program itself
  - Find XTI for an object (using RTTI) or for a class (by name)
- Read in XTI
  - make a modified copy
  - write XTI, C++, IDL, XML, …
- Read XTI from two sources
  - Compare, merge, …
- Add new class or operation to XTI
  - but not use it in the current program
  - Output code, compile, link, and use
- Look at XTI, see what functions are available
  - But not call them
  - Build function table, link it, then call

Relationship with platform services



- XTI can
  - be common interface to common services
    - Minimizing a program’s platform dependencies
  - extend platform services to cover Standard C++
    - Platforms often support “common language facilities” only
  - support platform-specific facilities through optional extensions to XTI
    - potential for thin layer common interfaces to non-universal services
    - Hard to do

Generation of inter-object communication code



Simple code example

```
Program prog(cin);
Scope& p = prog.global_scope();

for (Scope::iterator pp=p.begin(); pp!=p.end(); ++pp)
        cout << pp->name() << '\n';        // name of global

for (Scope::iterator pp=p.begin(); pp!=p.end(); ++pp)        // tree climbing
    if (pp->has_type())
        cout<< pp->name() << " : " << pp->get_type().xti_name() << '\n';
    else
        cout<< pp->name() << "\n";

class My_visitor : public XTI_visitor { /* … */ };
p.accept(vis);
```

Annotations

```
class Dcl_impl : public virtual Dcl {
    /*   base for Dcl_*_impl classes
         never instantiated by itself, type_impl() is virtual to ensure that
         (even though not every Dcl_*_impl class can meaningfully define it)
         not a XTI_obj
    */
    Name_impl* n;          // a declaration knows its name
    Dcl_properties mod;    // access, etc.
    Scope_impl* anno;      // #(name,value) annotations
public:
    // …
};
```

Overloading

- 1st idea, fct_type_list
  - "A function can have several types"
  - Doesn't work
    - type + non-type overload
- 2nd idea, type_list
  - Doesn't work
    - access control
    - Inline, mutable, static, etc., apply to declaration, not type
- 3rd idea, Dcl_set
  - Can handle undeclared names
  - Can handle arbitrary overloads
    - e.g. function and class, function and object
  - Makes it necessary to change of type after declaration
    - E.g., Dcl_fct to Dcl_set

How many classes?

- How to represent declarations
  - Alternatives
    - Dcl_ordinary
    - Dcl_type, Dcl_object, Dcl_fct, … (not distinguished in C++ grammar)
  - Most application code wants to distinguish: use several classes
    - User writes N virtual functions
- How to represent a fundamental type
  - Alternatives
    - Fundamental
    - Int, Double, Char, …
  - Most application code isn't interested in which fundamental type is presented: use a single "Fundamental" class
    - User writes one virtual function containing tests
    - Const/volatile affected that decision

Concrete XTI interface

- How to allow updates
  - Get/set
    - ugly, inherently unstructured, and error-prone
  - Get + whole-object assignment
    - Doesn't work with derived classes – slicing.
  - Return pointers to interesting members + a few set functions

  - Co-variant return defeated by **const**

Program

- A **Program** object
  - is the root of an XTI data structure
  - is responsible for deleting XTI objects that it owns
- How does a user get a first Program object
  - Cannot be abstract
    - Factory object with default (general, ugly, works)
    - C-style implementation supplied new_program()
  - Needs to be initialized from somewhere
- A program can contain several **Program** objects
  - So you can write programs comparing types from different sources, e.g. do global consistency checks

Call interface

- Requires run-time support
  - E.g. function tables generated by some other XTI program
  - Unsatisfactory design/notation:

```
void user(const char* f, string ff, const Function& fff)
{
    int i1 = fct_call(f,2);          // return type requires "magic"
    int i2 = fct_call(ff,2);
    int i3 = fct_call(fff,2);
    int i4 = fff(2);                 // overload operator() for Function
}
```

Query interface

- XTI has a minimal interface
  - Logically complete
  - Needs support library for convenience
- Programs can be written as sets of overloaded functions plus a simple loop
  - External polymorphism essential for extensibility
- Define composable function objects for use with algorithms

```
class Is_public { /* …*/ };
class Is_virtual { /* … */ };
Scope& s;
Scope_ref s2 =
    extract(s,compose2(logical_and<bool>,Is_public,Is_virtual)());
```

- Use lambdas to simplify notation?

```
Lambda x;
Scope_ref s3 = extract(s,is_public(x)&&is_virtual(x));
```

Random observations

- Strive for simplicity and symmetry
  - You can’t optimize every dimension at once
    - Settle for balance
- Design tools
  - Colleagues
  - Simple diagrams
  - Compiler
  - Parser generator (for checking only)

Current status

- Specification
  - Still mutating
  - What information should be available as XTI? Why? How?
- XTI class hierarchy
  - Still mutating (982+588 lines (incl. comments), about 70 classes)
- GCC produces hard-to-read type information
  - Support patch in 3.0
- GCC output to XPR generator
  - Incomplete (approximately 1000 lines)
- XPR reader
  - Incomplete (approximately 1000 lines)
- Call interface
  - Minor experiments
- Query interface
  - Back of the envelope draft